UM0304
User manual

STR91x DSP library (DSPLIB)

Introduction

This manual presents a library of ARM assembly source code modules for digital signal processing (DSP) applications such as infinite impulse response (IIR) filter, finite impulse response (FIR) filter and fast Fourier transform (FFT) applicable for a range of DSP applications including VSLP vocoder. These assembly source code modules are presented for ARM mode and have been tested in an ARM9E-based STR91x platform.

In addition, the assembly source code modules have been tested in an IAR Workbench environment as well, but STMicroelectronics cannot guarantee that these assembly source code modules will be flawless for all applications.

The algorithm modules are presented "as is with no warranty".

Contents

1 Definitions and related documents

1.1 Acronyms and terminology

Table 1. Definition of acronyms and terms

Term	Definition
ARM	ARM Core
ARMA	Auto Regressive Moving Average
DSP	Digital Signal Processing
DSPLIB	Digital Signal Processing Library
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
IIR	Infinite Impulse Response
LTI	Linear Time-Invariant
MCU	Microcontroller Unit
STR91x	STR91x family of MCUs from STMicroelectronics
VSELP	Vector-Sum Excited Linear Prediction

1.2 ARM and Thumb

The Thumb set consists of 16-bit instructions that act as a compact, shorthand subset of the 32-bit ARM instructions. Every Thumb instruction could be executed via an equivalent 32-bit ARM instruction. However, not all ARM instructions are available in the Thumb subset. For example, there's no way to access status or coprocessor registers in Thumb. Also, some functions that can be accomplished in a single ARM instruction can only be accomplished with a sequence of Thumb instructions.

Thumb compatible processor can operate in ARM or Thumb state. Some method is needed to switch the processor from executing instructions in one state to executing in the other. This is provided by the Branch Exchange instruction, versions of which exist both in the ARM and Thumb instruction sets. Both of these perform a branch by copying the contents of general register Rn into the program counter causing a pipeline flush and refill from address specified in Rn. Thus, BX is absolute rather than PC-relative.

```
In ARM state the format is:

       BX{<cond>} Rn

In Thumb state the format is:

       BX Rn
```

All ARM instructions are word-aligned, and all Thumb instructions are half-word aligned. Therefore, the least significant bit in Rn can always be considered to be zero. The

processor can actually use this bit to determine if the instruction jumped to should be executed in Thumb or ARM state:

- If bit 0 set then execute in Thumb state
- If bit 0 clear then execute in ARM state

Thumb was defined for two main reasons:

1. Better code density, as the instructions are half the size of ARM instructions (although some ARM instructions require two Thumb instructions for the same effect). You would have to compile the application for ARM and Thumb and see what gives the best result.
2. Better performance from narrow memory, as instruction fetches from smaller memory (ie: 8-bit or 16-bit) will be reduced in Thumb mode.

1.3 References

1. Sanjit K. Mitra, "Digital Signal Processing - A Computer Based Approach", McGraw Hill, Third Edition 2006.
2. R. Deka and J. G. Gardiner, "On the Fundamentals of Digital Signal Processing Micros," Journal of Microcomputer Applications, Vol 17 No 1, pp 101-135, U K, January 1994.
3. E. Oran Brigham, "The Fast Fourier Transform and its Applications", ISBN 0-13-307547-8, Prentice-Hall International Editions, 1988.
4. C.S. Burrus, "Unscrambling for fast DFT algorithms", IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-36(7), 1086-1089, July 1988.
5. C. S. Burrus and T.W. Parks, "DFT/FFT and Convolution Algorithms - Theory and Implementation", J. Wiley, 1985.

2 IIR ARMA 16-bit filter

2.1 Description

The IIR filter function performs an ARMA filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients respectively for ny output samples. This is a 16-bit fixed-point implementation. Reference [1] may be used to explore more about IIR filters.

2.2 Function

```
void iirarma_arm9e(void *y, void *x, short *h2, short *h1, int ny);
```

2.3 Arguments and variables

- Output array vector y[ny+4] (used in actual computation. first four elements must have the previous outputs. Thus the first output starts y[4] when ny = 0)
- Input array vector x[ny+4]
- Moving-average filter coefficients vector h2[5]
- Auto-regressive filter coefficients h1[5] and h1[0] are not used
- Number of output samples ny is a multiple of 4 and must be ≥ 8

2.3.1 Calling the function from C

Set the arguments and variables as appropriate in C main program and call the function.

In the STR91x DSPLIB this is set in the 91x_dsp.h header file.

2.3.2 Calling the function from assembly

Use the following registers equivalent to arguments/variables.

- R0 to the address of output vector y[]
- R1 to the address of input vector x[]
- R2 to the address of coefficient vector h2[]
- R3 to the address of coefficient vector h1[]
- R12 to the address of ny

Then use the BL instruction to call the function.

2.4 Algorithm

Following is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

C equivalent IIR ARMA algorithm

```
iir_arma_arm9e(short *y, short *x, short *h2, short *h1, int ny)
{
  int i, j;
  int sum;

  for (i=0; i<ny; i++){
    sum = h2[0] * x[4+i];
    for (j = 1; j <= 4; j++)
      sum += h2[j]*x[4+i-j]-h1[j]*y[4+i-j];
    y[4+i] = (sum >> 15);
  }
}
```

2.5 Assembly code

The assembly code for this IIR ARMA algorithm is validated in STR91x environment for ARM966E-S. The assembly code is for 16-bit fixed point applications. The assembly source code module may be found in DSPLIB.

2.6 Requirements

The assembly module is little endian and written for ARM9E in ARM mode with stack aligned to 8 bytes.

The module is for 16-bit fixed point applications where input data is expected to be 16-bit fixed point and it will produce 16-bit fixed-point outpute.

A caller needs to organize an input vector x[] for filter input and output vector y[] for filter output then supply the filter coefficients h1[] as well as h2[]. The vector length for x[] and y[] needs to follow the following guidelines.

- ny is multiple of 4 and greater than or equal to 8
- Input data array x[ ] contains ny + 4 input samples to produce ny output samples.

2.7 Implementation

- The output vector y[] contains ny + 4 locations, but its first 4 data are not used, i.e., first output is y[4] while ny = 0.
- The inner loop that iterated through the filter coefficients is completely unrolled.
- The code is little ENDIAN.
- The code is interrupt-tolerant but not interruptible.

2.8 Benchmarking

The benchmarking for this filter function is done using Chameleon Trace while the filter function is running in STR91x engine with ARM966E-S. For the Chameleon Trace in ETM Configuration Trace Port Size is set to 8 bit and Trace Port Mode is set to Normal, Half-rate clocking, while the STR91x engine runs at 96 MHz.

Table 2. Benchmarking of IIR ARMA algorithm module

Number of taps	Cycle count	Microseconds
72	6544	68.167
48	4396	45.792
24	2248	23.417

3 Block FIR 16-bit filter

3.1 Description

This function computes a direct-form real FIR filter using the coefficients stored in vector h[] using a simple Block FIR technique, and moves delay line. The input sequence needs to be start with (T -1) zeros. This is a 16-bit fixed-point implementation. Reference [*1*] may be used to explore more about FIR filters.

The user needs to organize the output buffer for y[N] and needs to design the filter taps h[T] and organize the coefficients in reverse order. The users also need to scale the input x[M] and filter taps h[T] to avoid overflow.

This function may be called by C or Assembly function(s).

3.2 Function

```
typedef struct coef{
  short *h;
  unsigned int m;
}nh;
 int fir_16by16_arm9e(int y[], short x[],  sx *p, int N);


    nh p;
    p.h = h;
    p.M = T;
```

3.3 Arguments and variables

- h[T] = filter coefficient vector with T number of taps, an integer multiple of 6
- T = number of filter coefficients (taps), an integer multiple of 6
- N = filter length or vector length for output
- x[M] = filter input vector with total M samples, i.e., (M = N + T - 1)
- y[N] = filter output vector with total N samples

3.3.1 Calling the function from C

Set the arguments and variables as appropriate in C main program and call the function.

3.3.2 Calling the function from Assembly

Use the following registers equivalent to arguments/variables.

- Move N to R3, N is output filter length or output vector length, a multiple of 6
- Move SP to R2, comparable &p as in C function above h[T] = filter coefficient vector with T number of taps, T is an integer multiple of 6
- Set R1 to the address of x, comparable as in x[M] filter input vector with total M samples, i.e., (M = N + T - 1)
- Set R0 to the address of y, comparable as in y[N] filter output vector with total N samples

Then use BL instruction to call the function.

3.4 Algorithm

Following is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

Block FIR Filter Reference C Source

```
void fir_16by16_arm9e(short input, int nt, short h[], short z[])
{
    int j;
    short acc;
/* store input at the beginning of the delay line */
    z[0] = input;
/* calc FIR */
    acc = 0;
    for (j = 0; j < nt; j++) {
        acc += h[j] * z[j];
    }
/* shift delay line */
    for (j = nt - 2; j >= 0; j--) {
        z[j + 1] = z[j];
    }
    return acc;
}
```

3.5 Assembly code

The assembly code for this block FIR 16-bit fixed-point filter algorithm is validated in STR91x environment for ARM966E-S. The assembly code is for 16-bit fixed point applications, and

the filter coefficients are a multiple of 6 - more information is available in the source code. The source code module may be found in DSPLIB.

3.6 Requirements

A caller needs to organize an input vectors say x[] and output vector say y[] for filter. The caller needs to design the filter coefficients using any of the windowing functions such as Hamming, Hanning, and Blackman etc as required by the filter response.

Then upon designing the coefficients, reverse the order of the coefficients. The number of coefficients, T, must be an integer multiple of 6.

3.7 Implementation

- The filter needs T number of coefficients and T is multiple of 6
- The output vector y[N] is such that N equals (2 * T)
- The input vector x[M] is such that M equals (N + T - 1)
- The code is little ENDIAN
- The code is interrupt-tolerant but not interruptible

3.8 Benchmarking

The benchmarking for this filter function is done using Chameleon Trace while the filter function is running in STR91x engine with ARM966E-S. For the Chameleon Trace in ETM Configuration Trace Port Size is set to 8 bit and Trace Port Mode is set to Normal, Half-rate clocking, while the STR91x engine runs at 96 MHz.

Table 3. Benchmarking of Block FIR algorithm module

Number of taps	Cycle count	Microseconds
72	15063	156.854
48	6842	71.271
24	1813	18.885

4 Complex 16-bit radix-4 FFT

4.1 Description

The DFT X[k] of a complex sequence x[n] of length N is calculated by:

X[k] = S n=0 N-1 x[n]*exp(-j2pnk/N) for k=0, 1,2 0.N-1

A direct calculation of N complex values of X[k] will require 4N2 multiplications and 4N(N-1) additions given the trigonometric function values. For example, if N=128, 65536 multiplications and 65024 additions are required.

In this report, a radix-4 FFT algorithm is implemented. More details about FFT algorithms including radix-4 are found in references [3] [4] and [5].

4.2 Algorithm

The algorithm works as follows:

Block 1: Bit reverse

Copy the data from the input buffer to the output buffer in bit-reversed order. It is an integer between 0 and N-1, with binary representation as shown, for the bit reversal of k, we will write. So we perform the equivalent of the following loop:

```
int k, khat, bit;
for (k=0, khat=0; k<N; k++) {
  X[khat]=x[k];
  for (bit=N/2; (khat & bit)!=0; bit >>=1) khat ^= bit;
khat ^= bit; /* finish incrementing khat */
}
```

The bit loop inside increments khat as a bit reversed number. If the input buffers and output buffers are equal (X=x), more care needs to be taken to do the action in place, but it can be performed faster as fewer elements need to be moved. Putting the elements in bit reversed order has the effect of grouping together all the Ys and Zs of the previous subsection so no further rearrangement needs to be done.

Block 2: Kernel

Perform the equivalent of the following loop:

- denotes complex multiplication

```
# define pi 3.14159265358


for (n=2; n<=N; n <<= 1) {
  w=2*pi/n;
  for (m=0; m<N; m+=n) {
    for (k=0; k<n/2; k++) {
      y=X[m+k];
      z=X[m+k+n/2] * exp(-ikw);
      X[m+k]=(y+z)/2;
      X[m+k+n/2]=(y-z)/2;
    }
  }
}
```

After n=2 we have performed N/2 two-element FFTs, positioned at offsets m=0, 2, 4, ... N-2.
After n=4 we have performed N/4 four-element FFTs positioned at offset m=0, 4, 8, … N-4.
After n=N we are left with the answer to the main FFT in the buffer X.

```
k bN 1 - ... b2
b1
b0
()?= 2
```

Because each pass of the FFT divides the data by 4 (radix4 passes) there is no bit growth. Four cases were tested as shown in *Table 4*.

Table 4. Bit growth of FFT routine

Case name	Description	Input amplitude	Output bin amplitude
Single sine max neg.	Single sine at 1600; Fs =25600; 1 complete waveform in 16 samples;	0x8001	2 bins of 0xC000 (+/- 0x2)
Single sine max pos.	Single sine at 1600; Fs =25600; 1 complete waveform in 16 samples;	0x7FFF	2 bins of 0x4000 (+/- 0x2)

Table 4. Bit growth of FFT routine

Case name	Description	Input amplitude	Output bin amplitude
Dual sine max neg.	Two sines at 800, 1200; Fs =25600;	0x8001	4 bins of 0xE000 (+/- 0x2)
Dual sine max pos.	Two sines at 800, 1200; Fs =25600;	0x7FFF	4 bins of 0x2000 (+/- 0x2)

4.3 Arguments and variables

The algorithm takes three values on input:

- A pointer x to the input data consisting of N complex numbers x[0], ..., x[N-1]. Each complex number is a 32-bit integer containing the 16-bit real part followed by the 16-bit imaginary part.
- A pointer y to the output buffer for the transformed array to be stored. This buffer is the same in size as the input buffer x, with same order as in input buffer for the real part as well as the imaginary part.
- An integer nBin giving the base 2 logarithm of the number of points N in the Fourier Transform.

4.4 Function

```
extern void cr4fft1k16_arm9e(void *pssOUT, void *pssIN, int nBin);
```

The input to the function is an array of complex data with each even index the real part and imaginary part being in odd index, e.g., x[0] real part and it imaginary part x[1]; x[2] real part and it imaginary part x[3]; and so on … Similarly, the output is also in the same order.

The block outline of this FFT algorithm is shown in *Table 5*.

Table 5. Block outline of FFT routine

Name	Function	Implementation
save_context	Save all registers used by the routine	
d_main	Main loop; transfer 16 bytes per iteration	single loop
d_swit	Second level switch: transfer 4, 8 or 12 bytes	linear routine
d_switch	Third level switch: transfer 1, 2 or 3 bytes	switch case
restore_context	Restore all registers	

4.4.1 Calling the FFT function from C

To call the FFT routine from C, include both 91x_dsp.c and 91x_dsp.h which declares the type structure and the function FFT as:

```
extern void cr4fft1k16_arm9e(void *pssOUT, void *pssIN, int nBin);
```

Then to call the function from main use the following in main program as global:

```
struct complex_fft *test_fft = NULL;
```

and call the function using

```
test_fft->calc_cr4fft_1k(y, x, N);
```

The function can also be called directly using simple function call by main program. Note the following points:

- The buffers x and y may coincide. In this case the FFT is said to be done in place rather than out of place.
- To prevent overflow within the algorithm, the real and imaginary values in the array x should be sign extended 16-bit quantities (between -32768 and +32767).
- The values of N allowed are 64,256, 1024.

4.4.2 Calling the FFT function from assembly

To call the FFT routine from assembler, import the symbol FFT and set up the registers as follows:

- R0 to the address of the input buffer x
- R1 to the address of the output buffer y
- R2 to N

Call the FFT routine using BL FFT. On exit R0 will contain the exit code (0 if successful), R1-R3 and R14 will have been corrupted and R4-R13 preserved.

4.5 The FFT function characteristics

Table 6. The FFT function characteristics

Feature	Description	Note
Name	Complex FFT	
Algorithm choices	Radix4, DIT, Bit reverse in input, Not in place	
Number of points	64 or 256 or 1024	
Input data size	2 x 16-bit	
Input data organization	imag_real (little endian) such as real = address imag =address +2	
Input data order	Linear (bit reverse is done during the first pass; this is transparent to the user)	
Computation radix	Radix 4 (Burrus Parks)	
Overflow prevention	for every pass (radix 4) the data is right shifted by 2 so that the resulting value does not overflow;	see normalized note
Saturate on overflow	no	
Rounding	none	
Output data size	2 x 16-bit	

Table 6. The FFT function characteristics

Feature	Description	Note
Output data organization	imag_real (little endian) such as real = address imag =address +2	
Output data order	Linear	
Coefficient size	16-bit	
Coefficient organization	imag_real (little endian) such as real = address imag =address +2	
Coefficient computation method	table lookup : sine table	
Coefficient Table size	4080 bytes; = (1024 + 256 + 64+ 16) *4 *3/4	

The STR91x implementation of the bit reversal stage is straightforward and contained between the labels FFT and FFTSTART. The second stage begins at the label FFTSTART.

4.6 Performance benchmarking

We start by listing the performance for the speed optimized algorithm. Timings are given for the STR91x running at 96 MHz. The code size for the optimized algorithm is 1592+ N/2 bytes (including the lookup-table) and the data size is 64 bytes (not including the input and output buffers).

Table 7. Benchmarking of Radix-4 Complex FFT Algorithm: STR91x 96 MHz.

Radix-4 complex FFT	Operation mode	Cycle count	Microseconds
64 Point	Program in Flash & Data in SRAM	2701	28.135
	Program & Data in SRAM	3432	35.750
	Program & Data in Flash	3705	38.594
256 Point	Program in Flash & Data in SRAM	13740	143.125
	Program & Data in SRAM	18079	188.323
	Program & Data in Flash	19908	207.375
1024 Point	Program in Flash & Data in SRAM	68534	713.896
	Program & Data in SRAM	90422	941.896
	Program & Data in Flash	101151	1053.656

All these results were obtained on a STR910-EVAL best case configuration for speed. The clock is 96 MHz; the number of wait states is minimum for Flash and for SRAM. The code is run from Flash.

4.7 Fixed-point error benchmarking

Table 8 shows fixed-point error analysis of FFT function (ARM966E-S) with respect to floating-point and fixed-point C function, and amplitude, for a 16-point FFT.

Table 8. The FFT function fixed-point error benchmarking

Real Part			Imaginary Part			Amplitude
ARM966E-S	Floating point	Fixed-point C	ARM966E-S	Floating point	Fixed-point C	
-5552	-5549.4375	-5550	-5552	-5549.4375	-5550	7848.089844
21318	21320.33984	21319	-5841	-5840.29541	-5840	22105.78906
6297	6299.319336	6298	-1503	-1501.481567	-1502	6475.791016
4100	4101.071289	4100	-402	-401.039703	-401	4120.633301
3146	3146.563477	3146	208	208.557663	208	3153.467529
2573	2574.675781	2574	632	631.753662	631	2651.050293
2165	2166.108887	2165	967	967.559753	967	2372.382568
1836	1836.466064	1836	1262	1262.519897	1262	2228.578857
1543	1544.313599	1544	1543	1544.311401	1544	2183.987549
1261	1262.530396	1262	1836	1836.464233	1836	2228.583496
966	967.565186	967	2164	2166.102539	2165	2372.37915
631	631.76416	631	2574	2574.674316	2574	2651.051514
208	208.583069	208	3146	3146.570068	3146	3153.47583
-402	-401.032806	-401	4101	4101.062988	4100	4120.624512
-1501	-1501.468384	-1502	6298	6299.319824	6298	6475.788574
-5840	-5840.282227	-5840	21320	21320.34766	21319	22105.79297

5 Revision history

Table 9. Document revision history

Date	Revision	Changes
18-Jan-2007	1	Initial release
22-Jan-2007	2	Added references 3, 4 and 5 in *Section 1.3 on page 5* Updated *Section 4.1 on page 12*
09-Jun-2008	3	Removed references to obsolete products.